Automatic System Testing of Programs without Test Oracles

Christian Murphy, Kuang Shen, Gail Kaiser
Dept. of Computer Science
Columbia University
New York NY 10027
{cmurphy, ks2555, kaiser}@cs.columbia.edu

ABSTRACT

Metamorphic testing has been shown to be a simple yet effective technique in addressing the quality assurance of applications that do not have test oracles, *i.e.*, for which it is difficult or impossible to know what the correct output should be for arbitrary input. In metamorphic testing, existing test case input is modified to produce new test cases in such a manner that, when given the new input, the application should produce an output that can be easily be computed based on the original output. That is, if input x produces output $f(x)$, then we create input x' such that we can predict $f(x')$ based on $f(x)$; if the application does not produce the expected output, then a defect must exist, and either $f(x)$ or $f(x')$ (or both) is wrong.

In practice, however, metamorphic testing can be a manually intensive technique for all but the simplest cases. The transformation of input data can be laborious for large data sets, or practically impossible for input that is not in human-readable format. Similarly, comparing the outputs can be error-prone for large result sets, especially when slight variations in the results are not actually indicative of errors (*i.e.*, are false positives), for instance when there is non-determinism in the application and multiple outputs can be considered correct.

In this paper, we present an approach called *Automated Metamorphic System Testing*. This involves the automation of metamorphic testing at the system level by checking that the metamorphic properties of the entire application hold after its execution. The tester is able to easily set up and conduct metamorphic tests with little manual intervention, and testing can continue in the field with minimal impact on the user. Additionally, we present an approach called *Heuristic Metamorphic Testing* which seeks to reduce false positives and address some cases of non-determinism. We also describe an implementation framework called *Amsterdam*, and present the results of empirical studies in which we demonstrate the effectiveness of the technique on real-world programs without test oracles.

ISSTA 2009 Chicago, IL USA

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Software Testing, Oracle Problem, Metamorphic Testing

1. INTRODUCTION

Assuring the quality of applications such as those in the fields of scientific calculations, optimizations, machine learning, *etc.* presents a challenge because conventional software testing processes do not always apply: in particular, it is difficult to detect subtle errors, faults, defects or anomalies in many applications in these domains because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. The general class of software systems with no reliable test oracle available is sometimes known as "non-testable programs" [37]. These applications fall into a category of software that Weyuker describes as *"Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known"* [37].

One approach to testing such applications is to use a "pseudo-oracle" [10], in which multiple implementations of an algorithm process an input and the results are compared; if the results are not the same, then one or both of the implementations contains a defect. This is not always feasible, though, since multiple implementations may not exist, or they may have been created by the same developers, or by groups of developers who are prone to making the same types of mistakes [20].

In the absence of multiple implementations, metamorphic testing [6] can be used to produce a similar effect. Metamorphic testing is designed as a general technique for creating follow-up test cases based on existing ones, particularly those that have not revealed any failure, in order to try to find uncovered flaws. Instead of being an approach for test case selection, it is a methodology of reusing input test data to create additional test cases whose outputs can be predicted. In metamorphic testing, if input x produces an output $f(x)$, the function's so-called "metamorphic properties" can then be used to guide the creation of a transformation

function t, which can then be applied to the input to produce $t(x)$; this transformation then allows us to predict the output $f(t(x))$, based on the (already known) value of $f(x)$. If the output is not as expected, then a defect must exist. Of course, this can only show the existence of defects and cannot demonstrate their absence, since the correct output cannot be known in advance (and even if the outputs are as expected, both could be incorrect), but metamorphic testing provides a powerful technique to reveal defects in such non-testable programs by use of a built-in pseudo-oracle.

In practice, however, metamorphic testing can be a manually intensive technique for all but the simplest cases. The transformation of input data can be laborious for large data sets, or practically impossible for input that is not in human-readable format. Similarly, comparing the outputs can be error-prone for large result sets, especially when slight variations in the results are not actually indicative of errors (*i.e.* false positives), or when there is non-determinism in the results. Another issue is where the initial input x comes from; manual creation of test input is challenging for non-trivial cases, and random input data may not reveal certain defects [23]. Last, it is time consuming to manually set up the input data, re-run the program, and compare the results.

Here, we present a solution to these limitations of metamorphic testing, describe an implementation, and measure its effectiveness. This paper makes four contributions:

1. We first describe an approach called *Automated Metamorphic System Testing*, which involves automating metamorphic testing at the system level by treating the application as a black box and checking that the metamorphic properties of the entire application hold after its execution. In the development/testing environment, the tester is able to easily set up and conduct metamorphic tests with little manual intervention; this will not require the tester to have access to the source code, but only to know the system's metamorphic properties. Once the testing is completed in the development environment, the approach will also allow for metamorphic testing to be conducted automatically in the deployment environment.

2. The testing approach is supported by an implementation framework called *Amsterdam*. This testing framework automates the process by which program input data is modified, multiple executions of the application with its different inputs are run in parallel, and the outputs of the executions are compared to check that the metamorphic properties are satisfied. When done in the deployment environment, a "sandbox" must be created such that the user sees only the results of the main (original) execution, and not others that are only for testing purposes.

3. We also present an approach called *Heuristic Metamorphic Testing*. In many cases, the limitations of floating point calculations may cause results of computations to appear to be different (thus indicating a defect), even though no defect exists. Additionally, non-deterministic applications make it difficult to accurately predict what the expected outputs should be during metamorphic testing. By setting thresholds and allowing for application-specific definitions of "close enough", we can reduce the number of false positives and address some cases of non-determinism.

4. Finally, we provide the results of empirical studies conducted on real-world non-testable programs (from the domain of machine learning) to demonstrate the effectiveness of our techniques.

2. BACKGROUND

Our work to date has primarily focused on the quality assurance of machine learning applications. As these types of applications become more and more prevalent in various aspects of everyday life [21], it is clear that the dependability of machine learning software takes on increasing importance. The majority of the research effort in the domain of machine learning focuses on building more accurate models that can better achieve the goal of automated learning from the real world. However, to date very little work has been done on assuring the correctness of the software applications that perform machine learning. Formal proofs of an algorithm's optimal quality do not guarantee that an application implements or uses the algorithm correctly, and thus software testing is necessary.

Previously we have applied metamorphic testing as part of a combined approach to testing machine learning applications [22]. Here, we focus on improving and measuring the metamorphic testing technique itself, and apply our approach to applications in the domain of machine learning, since there are extra challenges present in testing "non-testable programs". However, the techniques presented in this paper are applicable to other application domains as well, including those that do, in fact, have test oracles.

2.1 Metamorphic Testing Examples

A simple example (for expository purposes only, outside the domain of machine learning) of a function to which metamorphic testing could be applied would be one that calculates the standard deviation of a set of numbers. Certain transformations of the set would be expected to produce the same result. For instance, permuting the order of the elements should not affect the calculation; nor would multiplying each value by -1, since the devation from the mean would still be the same (think about the numbers being "flipped" around the zero on the number line).

Furthermore, other transformations will alter the output, but in a predictable way. For instance, if each value in the set were multipled by 2, then the standard deviation should be twice as much as that of the original set, since the values on the number line are just "stretched out" and their deviation from the mean becomes twice as great. Thus, given one set of numbers, we can create three more sets (one with the elements permuted, one with each multiplied by -1, and another with each multiplied by 2), and get a total of four test cases; moreover, given the output of only the first test case (even if we cannot know whether it was right or wrong), we can predict what the other three should be.

As a more complex example from the domain of machine learning, anomaly-based network intrusion detection systems build up a model of "normal" behavior based on what has previously been observed; this model may be created, for instance, according to the byte distribution of incoming network payloads (since the byte distribution in worms, viruses, *etc.* may deviate from that of normal network traffic [36]). When a new payload arrives, its byte distribution is then compared to that model, and anything deemed anomalous causes an alert. For a particular input, it may not be possi-

ble to know *a priori* whether it should raise an alert, since that is entirely dependent on the model. However, if while the program is running we take the new payload and randomly permute the order of its bytes, the result (anomalous or not) should be the same, since the model only concerns the distribution, not the order. If the result is not the same, then a defect must exist.

Clearly metamorphic testing can be very useful in the absence of an oracle: regardless of the values, if the different outputs for the different inputs are not as expected, then there must be a defect in the implementation. Although the use of these simple relationships for testing numerical functions is not unique to metamorphic testing (*e.g.*, testing based on algebraic properties [9] or programs that can check their work [4]), the approach can be used on a broader domain of any functions that display metamorphic properties. Additionally, metamorphic testing can treat the application under test as a black box, and does not require detailed understanding of the source code.

2.2 Limitations of Metamorphic Testing

Although it has been demonstrated (*e.g.*, in [25]) that it is possible to use metamorphic testing to reveal previously-unknown defects in applications without test oracles, the process by which the testing is conducted still has some limitations from a practical point of view.

For instance, the manual transformation of the input data can be laborious and error-prone, especially when the input consists of large tables of data, rather than just scalars or small sets. Machine learning applications, for example, can take input files of thousands or tens of thousands of rows of data; anything but the simplest transformations would need to be automated. On a similar note, input data that is not human-readable (for instance, binary files representing network traffic) cannot easily be manually modified and thus a tool is necessary. One-off scripts could be created, but to date there is no general framework that addresses different types of transformations and different types of input formats for purposes of metamorphic testing.

Additionally, the manual comparison of the program outputs can also cause problems. As with the input data, many applications for which metamorphic testing is appropriate produce large sets of output, and comparing them manually can be error-prone and tedious. Tools like "diff" are only useful if the results are expected to be exactly the same, but cannot be used if the relationship between the outputs is more complicated, for instance if the result of the modified input is a set of data that is equal to the result of the original input, but with each value multiplied by two.

The problem with comparing program outputs - regardless of how it is done - is exacerbated by the fact that imprecisions in floating point operations in digital computers could cause outputs to appear to deviate, even though the calculation is actually correct programatically; this could yield numerous false positives. Similarly, non-deterministic programs may yield outputs that are *expected* to deviate, and thus it may be impossible in practice to know whether the output is one that is predicted, given the non-determinism.

Last, in order to generate the test cases, metamorphic testing requires the initial input and output values. These could be generated using techniques for creating test input, but these techniques might miss some defects, since they might not consider a sufficient variety of potential system states or execution paths in the program. Some defects in such systems may only be found under certain application states that may not have been tested: for large, complex software systems, it is typically impossible in terms of time and cost to reliably test all possible system states before releasing the product into the field.

Thus, we require a strategy that addresses these limitations and improves the process by which metamorphic testing is conducted in practice.

3. RELATED WORK

Gotleib and Botella [12] coined the term "automated metamorphic testing" to describe how the process can be conducted automatically, but their work focuses more on the automatic creation of input data that would reveal violations of metamorphic properties, and not on automatically checking that those properties hold after execution. Also, they do not describe any mechanism for addressing performance concerns or for ensuring that the additional invocation of the function or the program is not seen by the user (*i.e.*, their approach was targeted at the development environment, whereas we target both the development environment and the deployment environment). Additionally, they only provided means for automating the testing of programs written in C; our Automated Metamorphic System Testing framework can be used with programs written in any language. And while we recognize that the automation of an existing manual process is not in itself a contribution, the work we present here introduces notions of parallelism and sandboxing to metamorphic testing, which have not previously been investigated.

Applying metamorphic testing to situations in which there is no test oracle has previously been studied by Chen *et al.* [7] [8]. However, this previous work did not consider the challenges of automating the process, but rather relied on a tester to manually perform the transformations and comparisons; in this work, we automate the transformation of the inputs and the comparison of the outputs, which both simplifies the process and makes it less error-prone.

Additionally, although some work has been done in investigating the use of metamorphic testing of non-deterministic applications [14], the "heuristic metamorphic testing" approach presented here is unique in considering how to reduce false positives.

Metamorphic properties are similar in some ways to algebraic specifications [9], though algebraic specifications often declare legal sequences of function calls that will produce a known result, typically within a given data structure (*e.g.* $pop(push(X)) == X$ in a Stack), but do not describe how an entire application should react when its input is changed. The runtime checking of algebraic specifications has been explored in [28] and [32], though neither work considered the particular issues that arise from testing without oracles. Even in the cases in which algebraic specifications or formal specification languages (such as Alloy [19], Z [2], *etc.*) are used to act as oracles, work to date has focused primarily on consistency checking of abstract data types [33] and has not sought to create oracles for applications that do not otherwise have them.

The implementation framework presented here extends our previous work in "in vivo testing" [24] in which software tests itself as it runs in the field, and adds to a list of automated testing tools such as Mercury WinRunner, Korat [5],

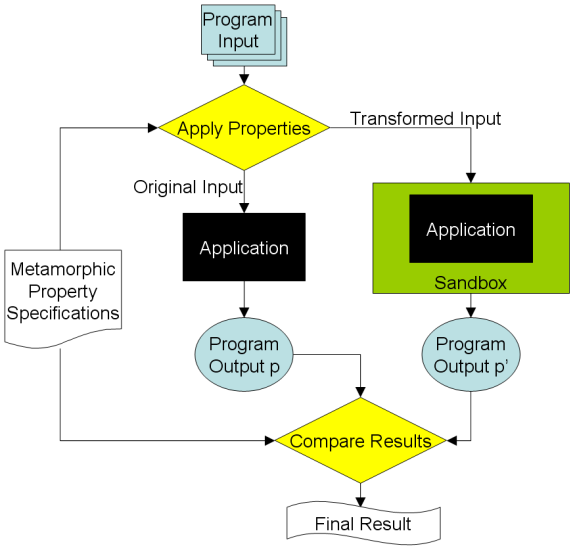

Figure 1: Model of Automated Metamorphic System Testing Framework

etc.; Amsterdam differentiates itself from these others by explicitly addressing the problems associated with testing applications without test oracles.

4. APPROACH

To address some of the limitations of metamorphic testing described above, we present a new technique called *Automated Metamorphic System Testing*. This automates the process by which program input data is modified, multiple executions of the application with its different inputs are run in parallel, and the outputs of the executions are compared to check that the metamorphic properties are satisfied. Aside from facilitating metamorphic testing in the development environment, this technique can also be used to continually test the application as it runs in the deployment environment, as well. This must be done in such a manner that the user only sees the results of the main (original) execution, and not from any of the others that are only for testing purposes.

4.1 Model

In the model of this approach (Figure 1), metamorphic properties of the application are specified by the tester and then are applied to the program input. The original input is fed into the application, which is treated completely as a black box; depending on the metamorphic property, a modified version of this input data may also be produced. That data is then fed into a separate invocation of the application, which executes in parallel but in a separate sandbox so that changes to files, screen output, *etc.* are not seen by the user. When the two invocations finish, their results are compared according to the specification; if the results are not as expected, a defect has been revealed. Although not reflected in Figure 1, it should be possible to execute more than two invocations of the program in parallel.

Note that the tester need not write any actual test code per se, but rather only needs to specify the metamorphic properties of the application. This can be done by the creator of the algorithm or by application designer, and does not assume intricate knowledge of the source code or other implementation details.

4.2 Implementation

This section describes the architecture of a framework, called *Amsterdam*, that will enable an application to be treated as a black box so that Automated Metamorphic System Testing can be performed without any modification to the code whatsoever. This framework also allows the application to be tested as it runs in the field, using real input data. As described above, multiple invocations of the application are run and their outputs are compared; however, the additional invocations must not affect the user and must run in a separate sandbox.

4.2.1 Assumptions

The framework currently assumes that the program under test can be invoked from the command line, system input comes from files, and output is either written to a file or to standard out (the screen). Though this may limit the generality of this framework, according to our preliminary investigations, these assumptions are typically not restrictive in applications in the domains of interest (particularly, machine learning). Additionally, when input comes from database tables, mouse clicks, keystrokes, incoming network traffic, *etc.*, an analogous unit testing approach such as the one described in [26] can be used instead, since that inserts code into the application, and that code can perform metamorphic testing at a more granular level.

4.2.2 Specifying Metamorphic Properties

The tester first specifies the metamorphic properties of the application. In our current implementation of the framework, this can be done in a text file using a syntax similar to plain English for some simple properties. For instance, if permuting the input to an application does not affect the output (*i.e.*, the resulting output is equal to the output of the initial test case), then the specification would simply be "if permute (input) then equal (output)". For more complex properties, an XML file is used for the specification (described below), and we are investigating other possibilities, such as using a formal specification language like Alloy [19], or a scripting language like Python or Perl. The examples in this section assume the specifications are written in XML (since the plain-English properties are pre-processed into XML files), though the ideas and principles will remain the same, regardless of the particular implementation.

The specification of a metamorphic property includes three parts: how to transform the input, how to execute the program (*e.g.*, the command to execute, setting any runtime options, *etc.*), and how to compare the outputs. Multiple metamorphic properties can be specified together in one file.

For **input transformation**, the tester can describe how to modify (if modification is needed at all) the entire data set or only certain parts, such as a particular row or column in a table of data. In [25], we identified six categories of metamorphic properties, and the framework supports out-of-the-box input modification functions to match each of these categories: adding a constant to numerical values; multiplying numerical values by a constant; permuting the order of the input data; reversing the order of the input data; removing part of the data; and, adding additional data.

For **program execution**, the tester needs to specify the command used to execute the program. The program is completely treated as a black box, so the particular implementation language does not matter, as long as the program is executable from the command line. Some metamorphic properties may call for different runtime options to be used for the different invocations; those would be specified here.

For **output comparison**, the tester describes what the expected output should be in terms of the original output. In the simplest case, the outputs would be expected to be exactly the same. In other cases, the same transformations described above (adding, multiplying, *etc.*) for the input may need to be applied to the output before checking for equality. Additionally, the framework also supports checking for *inequality* if a change to the input is expected to cause a change to the output, even if that output cannot be precisely predicted. Last, if the output is non-deterministic, heuristic metamorphic testing can be used, as described in Section 5.

```
<TESTDESCRIPTOR>
     <EXECUTION>java NaiveBayes @parameters</EXECUTION>
     <PARAMETERS>-t @input.training_data -d @output.model</PARAMETERS>
     <INPUT>
          <VAR TYPE="arff_file" NAME="training_data" />
     </INPUT>
     <OUTPUT>
          <VAR TYPE="text_file" NAME="model" />
     </OUTPUT>
     <POST_TEST>
          <BRANCH OPTION="main" />
          <BRANCH OPTION="parallel" NAME="test1">
               @op_permute(@input.training_data)
          </BRANCH>
          <PROPERTY>
               <ASSERT> @op_equal(@main.output.model, @test1.output.model) </ASSERT>
          </PROPERTY>
     </POST_TEST>
</TESTDESCRIPTOR>
```

Figure 2: Example of specification of metamorphic property for system-level testing

Figure 2 demonstrates an example of a metamorphic property for system testing as specified in an XML file. The input and output are given names, and the "post_test" specifies that there are to be two parallel executions and how to modify the inputs and compare the outputs. In particular, this file specifies that the NaiveBayes program (a machine learning classifier) has the property that, if the input ("training data") is perumuted, the output ("model") should still be the same. With minimal modification, the metamorphic properties specified in this XML file could also be applied to any other program that exhibits the same property.

If the framework does not support a specific transformation or comparison feature as required by the tester, functionality can be added by creating a separate component that can be invoked by the framework, according to a specific programming interface (currently implemented in Java). This would also allow the tester to automate the transformation of other input formats not currently supported by the tool, or to compare other output formats.

4.2.3 Configuration

After specifying the metamorphic properties, the tester then configures the Amsterdam framework to specify how the multiple invocations of the program should be executed. The framework is designed to be used in either the production environment (executed by the system's end users) or in the development environment (during pre-release testing). In the latter case, parallel execution of the additional invocations and/or the use of a separate sandbox may not be required; thus, parallelism and sandboxing can be disabled, which may ease the process of debugging (if, for instance, the tester wants to see the traces of debugging statements printed to standard out). The tester may also want to specify whether or not the additional invocations should run on separate processors or cores, if supported by the underlying hardware.

The configuration also includes declaring what action to take if a metamorphic test fails. Because the test can only complete once all invocations of the program have completed, it would be too late to "interrupt" program execution as soon as a defect is discovered, but the user can still be notified that the test revealed unexpected behavior, either through an entry in a log file or a pop-up window; in these cases, the user can know that the results of the program execution may be flawed. Likewise, the results of the test (including the input data that caused the failure) can be sent back to the development team for use in regression testing and program evolution.

4.2.4 Execution of Tests

When using Amsterdam to test an application as it runs in the field, it is assumed that the application vendor would ship the application including the configured testing framework as part of the software distribution. However, the customer organization using the software would not need to do anything special at all, and ideally would not even notice that the tests were running.

When the application is executed, the testing framework first creates temporary files to use for the additional invocations of the program and then invokes the original application with the command line arguments specified in the property specification, so that the startup delay of the framework is minimal from the user's perspective. While the application is running, Amsterdam then applies the specified transformations to the input files. This is done after invoking the original application because modifying large files can take a long time, and there is no need for the original application to wait. The framework provides out-of-the-box support for the transformation of four different file formats: XML, comma-separated value (CSV), an attribute/value pair format for "sparse" data, and the attribute-relation file format (ARFF). These file formats are commonly used in the domains of interest. Other file formats can be supported by building custom transformation components.

The framework then starts additional invocations with the newly-generated inputs. The sandbox for the parallel processeses is currently provided by creating isolated copies of all the files used by the test processes, and by redirecting screen output to a file so that the user does not see the results of the additional invocations of the program. To make the sandbox more robust, we have begun to investigate integration with a virtualization layer called a "pod" (PrOcess Domain) [29], which creates a virtual environment in which the process has its own view of the file system and process ID space and thus does not affect any other processes. At this time the framework sandbox does not include external entities such as the network or databases.

Once all processes are complete, the output files are then compared according to the specification of the metamorphic properties. If the output files are not as expected, then a defect has been detected, and the appropriate action can be taken according to the configuration.

4.3 Performance Overhead

To demonstrate that the Amsterdam framework incurs limited overhead on the application being tested, we conducted performance tests on a quad-core 3GHz CPU running Ubuntu 7.10. We tested four applications from the Weka 3.5.8 [38] toolkit for machine learning in Java. Our experiments showed that the performance impact on the main application process (the one seen by the user) comes only from the creation of the sandbox and copying the files for the input: this was measured at about 400ms for a 10MB input file. After that, all other test processes execute on separate cores and do not interfere with the original process (assuming that there are fewer test processes than cores, of course). Thus, the tests can be run with minimal performance impact from the user's perspective.

5. HEURISTIC METAMORPHIC TESTING

During the implementation of our Amsterdam framework, it became immediately apparent that false positives could be a problem if there were small deviations in the results of calculations that were expected to yield the same result. Additionally, many applications without test oracles rely on non-determinism, which limits the effectiveness of metamorphic testing since it makes it more difficult to predict the expected outputs. To address this, we introduce a technique called *heuristic metamorphic testing*, based on the concept of "heuristic test oracles" [18]. This variant of metamorphic testing permits slight differences in the outputs, in a meaningful way according to the application being tested.

5.1 Reducing False Positives

False positives are likely to come up in metamorphic testing whenever floating point calculations are involved. Consider the simple case of the sine function, and the metamorphic property $\sin(\alpha) = \sin(\alpha + 2\pi)$. In practice, a defect-free implementation may cause a failure of the metamorphic test, due to imprecision in floating point calculations and the representation of π. For instance, the Math.sin function in Java computes the sine of 6.02 radians and the sine of (6.02 + 2 * Math.PI) radians as having a difference on the order of 10^{-15}, which in most applications is probably negligible, but is not exactly the same when checking for equality; thus, a metamorphic property based on checking that the results are equal would lead to a false positive.

In heuristic metamorphic testing, output values that are "close enough" are considered equal, where the definition of "close enough" is dependent on the application or function being tested. For instance, when the output is numeric (as in the above example), a threshold can be set to check that the values are suitably close. More complex cases may call for heuristics to check for semantic similarity. For example, consider an application that executes in two phases, where the output of the first is used as part of the input to the second. That is, the final output of the program is the output of the second phase $S(F(x), y)$, where x is the initial input, $F(x)$ is the output from the first phase, and y is input used only in the second phase. Consider a case in which $F(x)$ is expected to equal $F(x')$, where x' is the input used in metamorphic testing. In heuristic metamorphic testing, even if $F(x)$ is not exactly equal to $F(x')$, the results may be considered "close enough" if $S(F(x), y)$ still equals $S(F(x'), y)$ (or is acceptably close).

The Amsterdam framework for Automated Metamorphic System Testing supports such techniques for considering such heuristics and setting thresholds in the comparison of outputs, with the intent of reducing false positives. However, in [39], it was argued that although the failure of a metamorphic test in such cases may not necessarily indicate a defect per se, it does reflect a deviation from expected behavior (albeit a very slight one), and thus it is useful to warn the user. Therefore, Amsterdam can be configured to generate such a warning if so desired.

5.2 Addressing Non-Determinism

Non-deterministic applications present a particular challenge to metamorphic testing because it may not always be possible to know what the expected output should be, and then check whether the new test output is as predicted. Statistical metamorphic testing [14] has been suggested as a solution in some cases, though it is somewhat limited to outputs that consist of a set of numbers whose statistical values, such as mean and variance, can be calculated. Statistical metamorphic testing is not necessarily applicable in the applications of interest presented here, for instance where the output could be a set in which the *ordering* is of prime concern, and the *values* of the set elements are not important.

For example, ranking algorithms in machine learning take sets of data and try to order the elements according to some learned relationship. Such algorithms may rely on randomness, for instance to permute the order of the input data set so that initial ordering does not influence the results, and then take the average ranking over a set of runs. Depending on these permutations, though, the final result may not be deterministic.

In heuristic metamorphic testing, the results of a ranking algorithm can still be compared using some basic metrics, such as the quality (measured using the Area Under the Curve, or AUC [16], a common metric for comparing machine learning results) for each ranking, the number of differences between the rankings (elements ranked differently), the Manhattan distance (sum of the absolute values of the differences in the rankings), and the Euclidean distance (in N-dimensional space). Another metric is the normalized Spearman Footrule Distance, which explains how similar the rankings are (1 means exactly the same, 0 means completely in the opposite order) [34].

Furthermore, in practice the user of the system may only be concerned with a small number of elements in the ranking, presumably selected from the top (or possibly the bottom) of the list. For a parameterized value X, it may be suitable to calculate the quality of only the top and bottom $X\%$ of each ranking, or calculate the "correspondence" between the top and bottom $X\%$ of both rankings, where the correspondence is simply the number of elements that appear in the top (or bottom) $X\%$ of both rankings, divided by the number of elements in the top (or bottom) $X\%$. These metrics, along with the other distance metrics described previously, can help decide whether a pair of rankings is similar in the ranges that are most important, and thus be used even when the result is non-deterministic.

Likewise, similar measurements can be used for calculating the similarity of results of machine learning classification algorithms, such as the number of elements with equivalent classifications, or the total number of elements in each class.

As with the examples aboved aimed at reducing the num-

ber of false positives, the Amsterdam framework supports heuristics related to non-determinism, but also can alert the user when the heuristics need to be used, so as to avoid possible false negatives.

6. EMPIRICAL STUDIES

To demonstrate the effectiveness of our technique and determine how well it can detect defects in software without test oracles, we conducted empirical studies on three real-world "non-testable programs" from the domain of machine learning. The first is the classification algorithm Support Vector Machines (SVM) [35], as implemented in the Weka [38] 3.5.8 toolkit for machine learning in Java. The second is C4.5 [31] release 8, which is also a classification algorithm but uses a decision tree. The last is the ranking algorithm MartiRank [13], developed by researchers at Columbia University's Center for Computational Learning Systems (CCLS).

6.1 Machine Learning Background

In supervised machine learning, data sets consist of a collection of *examples*, each of which has a number of *attribute* values and, in some cases, a *label*. The examples can be thought of as rows in a table, each of which represents one item from which to learn, and the attributes are the columns of the table. The label, if it exists, indicates how the example is categorized. Supervised machine learning applications execute in two phases. The first phase (called the *learning phase*) analyzes a set of *training data*; the result of this analysis is a *model* that attempts to make generalizations about how the attributes relate to the label. In the second phase (called the *classification phase*), the model is applied to another, previously-unseen data set (called the *testing data*) where the labels are unknown. In a classification algorithm, the system attempts to predict the label of each individual example; in a ranking algorithm, the output of this phase is a ranking such that, when the labels become known, it is intended that the highest valued labels are at or near the top of the ranking, with the lowest valued labels at or near the bottom.

6.1.1 Support Vector Machines

The Support Vector Machines (SVM) algorithm [35] is one of the more common classification algorithms used in real-world applications, ranging from facial recognition to computational biology [1]. In the learning phase, SVM treats each example from the training data as a vector of N dimensions (since it has N attributes), and attempts to segregate examples from different classes with a hyperplane of N-1 dimensions. In the learning phase, the goal is to find the hyperplane with the maximum margin (distance) between the "support vectors", which are the examples that lie closest to the surface of the hyperplane; the resulting hyperplane is the model. In the classification phase, examples in the testing data are classified according to which "side" of the hyperplane they fall on.

The Weka implementation of SVM uses the Sequential Minimal Optimization (SMO) technique [30], which breaks the large quadratic programming optimization problem into smaller problems that can be solved analytically and thus avoids a large matrix computation with limited loss of quality in the results.

6.1.2 C4.5

C4.5 [31] is a very popular algorithm for building decision trees, in which branches represent decisions based on attribute values and leaves represent how the example is to be classified. Like other decision tree classifiers, it takes advantage of the fact that each attribute in the training data can be used to make a decision that splits the data into smaller subsets. During the training phase, for each attribute, C4.5 measures how effective it is to split the data on a particular attribute value, and the attribute with the highest "information gain" (a measure of how well similar labels are grouped together) is the one used to make the decision. The algorithm then continues recursively on the smaller sublists. During classification, the tree is applied to each example, which is classified once it reaches a leaf of the tree.

6.1.3 MartiRank

MartiRank [13] is a ranking algorithm that is used as part of a prototype application for predicting electrical device failures: the examples in the data sets have labels of 0 ("negative example") or 1 ("positive example"), indicating whether the device failed during a particular time period.

In the learning phase, MartiRank executes a number of "rounds". In each round the set of training data is broken into sub-lists; there are N sub-lists in the Nth round, each containing $1/N$th of the total number of positive examples. For each sub-list, MartiRank sorts that segment by each attribute, ascending and descending, and chooses the attribute that gives the best "quality". The quality is assessed using a variant of the Area Under the Curve (AUC) [16] calculation that is adapted to ranking rather than binary classification. The model, then, describes for each round how to split the data set and on which attribute and direction to sort each segment for that round. In the second phase, MartiRank applies the segmentation and sorting rules from the model to the testing data set to produce the final ranking.

6.2 Methodology

In our experiments, we used mutation testing to systematically insert defects into the source code and then determined whether or not the mutants could be killed (*i.e.*, whether the defects could be detected) using our approach. Mutation testing has been shown to be a suitable technique for measuring the effectiveness of a test data suite or, in our case, a testing approach [3]. These mutations fell into three categories: (1) comparison operators were switched to their logical opposites, *e.g.* "less than" was switched to "greater than or equal"; (2) mathematical operators were switched to their opposites, *e.g.* addition was switched to subtraction; and (3) off-by-one errors were introduced for loop variables, array indices, and other calculations that required adjustment by one. Each variant that we created had exactly one mutation inserted.

To determine which mutations were suitable for our testing, the output of each variant was compared to the output of the application with no mutants, which was considered the "gold standard". To obtain this initial output, for SVM and C4.5, we used the "iris" data set from the UC-Irvine repository [27] (150 examples, five attributes); for MartiRank, we used a real-world data set used by the device failure application described above (10,000 examples, 119 attributes). If the outputs of the gold standard and the variant were the same, then we considered the mutation unsuitable for test-

Mutation	Mutants	Permute	Multiply	Add	Negate	Total
Comparison operators	30	17	2	0	0	17 (57%)
Math operators	24	13	0	11	16	18 (75%)
Off-by-one	31	27	0	7	9	31 (100%)
Total	**85**	**57**	**2**	**18**	**25**	**66 (77%)**

Table 1: Results of Mutation Testing for SVM

Mutation	Mutants	Permute	Multiply	Add	Negate	Total
Comparison operators	8	8	0	1	7	8 (100%)
Math operators	15	2	3	1	13	14 (93%)
Off-by-one	5	2	0	0	5	5 (100%)
Total	**28**	**12**	**3**	**2**	**25**	**27 (96%)**

Table 2: Results of Mutation Testing for C4.5

ing (since the mutation may not have been on the execution path, or may have been a "weak mutant" that did not affect the overall output). Additionally, if the mutation yielded a fatal error (crash), an infinite loop, or an output that was clearly wrong (for instance, being nonsensical to someone familiar with the application, or simply being blank), that variant was also discarded since our approach would not be needed to detect such defects.

Once we deteremined which variants could be used for our experiment, we conducted metamorphic testing using the following properties. Each property was verified with the "gold standard" version to make sure that the property was, in fact, expected to hold. Because of slight variations, the properties used for each application are listed separately:

For **SVM**: (1) Permuting the order of the examples in the training data should not affect the model; (2) Multiplying each attribute value in the training data by a positive constant (in our case, two) should not affect the model; (3) Adding a positive constant (in our case, one) to each attribute value in the training data should not affect the model; (4) Negating each attribute value in the training data, followed by negating each attribute value in the testing data, should result in the same classification.

For **C4.5**: (1) Permuting the order of the examples in the training data should not affect the model; (2) Multiplying each attribute value in the training data by a positive constant (in our case, two) should yield a model in which the values at each decision point have also been multiplied by two; (3) Adding a positive constant (in our case, one) to each attribute value in the training data should yield a model in which the values at each decision point have also been increased by one; (4) Negating each attribute value in the training data, followed by negating each attribute value in the testing data, should result in the same classification.

For **MartiRank**: (1) Permuting the order of the examples in the training data should not affect the model; (2) Multiplying each attribute value in the training data by a positive constant (in our case, two) should not affect the model; (3) Adding a positive constant (in our case, one) to each attribute value in the training data should not affect the model; (4) Negating each attribute value in the training data, followed by negating each attribute value in the testing data, should result in the same ranking.

For each variant, the application's metamorphic properties were specified using the Amsterdam testing framework and the tests were conducted to see whether the outputs were as expected (compared to the variant's original output). If not, then the mutant was considered to be killed, and the defect had been detected.

6.3 Findings

The goal of the experiment is to demonstrate that the metamorphic testing technique is effective in revealing the defects in these applications, by measuring what percentage of the mutants can be killed.

Tables 1, 2, and 3 summarize the results of our testing of SVM, C4.5, and MartiRank, respectively. In each table, we specify the type of mutation and the number of mutants that were suitable for use in the testing (*i.e.*, that resulted in a different output compared to the gold standard, and that did not produce an obvious error). We then list the number of mutants that were killed by metamorphic testing with the four different types of properties: permuting the input, adding a constant to each attribute, multiplying each attribute by a positive constant, and negating the attribute values. The last column shows the total number of distinct mutants killed by the tests (a mutant may be killed by multiple different metamorphic properties), and the overall percentage.

6.4 Discussion

For SVM, permuting the input was the most powerful metamorphic property in terms of revealing defects. Although we already demonstrated in [25] that permuting the input for SVM would cause this property to be violated, even in an implementation without defects, we avoided false positives in this case by using heuristic metamorphic testing and considering model values that were within 98% of each other to be considered "equal" (since this was the maximum margin of error in the gold standard version). Even with that buffer, 57 of the 85 mutants (67%) were killed.

Permuting the input was particularly effective in killing the off-by-one mutants in SVM (27 out of 31, or 87%). In these mutations, for-loops omitted either the first or last value in an array, thus the mathematical calculations would yield different results because different permutations meant

Mutation	Mutants	Permute	Multiply	Add	Negate	Total
Comparison operators	20	16	1	1	16	18 (90%)
Math operators	23	9	0	0	10	15 (65%)
Off-by-one	26	12	0	0	9	17 (65%)
Total	**69**	**37**	**1**	**1**	**35**	**50 (72%)**

Table 3: Results of Mutation Testing for MartiRank

that different elements were being left out. For instance, consider a function $f(A) = \sum_i A_i$, where A is an array of values. One would expect that permuting the order of the elements in A would not affect the result. But clearly if, say, the first element of A is not included in the sum, then permuting the elements will put a different one first, and thus the result will change, in violation of the metamorphic property.

On the other hand, because MartiRank is based on sorting, it follows that defects related to comparison operators would be most likely to be detected using the approach, particularly if the values are permuted or negated (these two properties each killed 16 of the 20 comparison operator mutations). Both types of metamorphic transformations affect the way in which the numbers are compared during sorting, and thus the results would be more likely to be different, indicating a defect.

For instance, consider a function to determine the maximum of three integers. One would expect that permuting the input should not affect the result, so that if $max(x, y, z) = z$, then $max(z, y, x)$ should still be z. However, if there were an error with one of the comparison operators, then permuting the order of the inputs could yield a different result, revealing the defect. Consider, for example, the erroneous implementation in Figure 3. Here, $max(2, 3, 4)$ would correctly return 4, but $max(4, 3, 2)$ would incorrectly return 2, indicating the defect. Although this (and the ones that follow) can be regarded as a trivial example, it demonstrates in a simple manner that the metamorphic property based on permutation is effective in applications like MartiRank that depend on comparisons.

```
1  max(a, b, c) {
2      if (a < b)
3          if (b < c) return c;
4          else return b;
5      else
6          if (a < c) return a;
7          else return c;
8  }
```

Figure 3: Mutated function to find maximum of three numbers with error on line 6

The testing approach was most effective for C4.5, particularly negating the input, which proved to be the most reliable means of killing the C4.5 mutants (25 of 28, or 89%), and was more or less equally effective for all three types of mutations. This is because the nodes of the decision tree contain clauses such as "if $attr_n > \alpha$ then class = C", where $attr_n$ is some attribute, α is some value, and C is the classification. If all the training data were negated, then the clause is expected to become "if $attr_n \leq$ -α then class = C", which requires both the comparison operator and the sign of α to be switched. However, in most of the cases, only one or the other was switched, so that in the classification phase, elements in the testing data were not correctly classified. Because C4.5 also involves calculations (to determine which splitting of data provides the best information gain), other mutations caused the value of α to be changed when the training data values were negated.

For all applications, the metamorphic property based on multiplication was not effective for revealing the types of defects that we had injected. The explanation is that for the operations that were changed by the mutations, they would still yield the same results because of the distributive properties of multiplication. Consider, for an example, a function $f(x, y) = x + y$. We would expect it to have the metamorphic property $f(2x, 2y) = 2f(x, y)$. Now consider a mutation of this function in which the plus sign has been replaced with a minus sign: $f'(x, y) = x - y$. Although there is an defect in the code, clearly the metamorphic property $f'(2x, 2y) = 2f'(x, y)$ still holds; thus, the metamorphic property based on multiplication would not show a violation.

However, this is not necessarily the case for addition, which does not have similar distributive properties. Consider the same function $f(x, y) = x + y$. We would expect it to have the metamorphic property $f(x + 2, y + 2) = x + 2 + y + 2 = f(x, y) + 4$. Now consider the same mutation of this function in which the plus sign has been replaced with a minus sign: $f'(x, y) = x - y$. Now the metamorphic property $f'(x + 2, y + 2) = f'(x, y) + 4$ no longer holds, because $f'(x + 2, y + 2) = x + 2 - (y + 2) = x - y = f'(x, y)$; thus, the metamorphic property based on addition would show a violation. As can be seen in Tables 1 and 3, though, this property is more effective in applications based on calculation and computation (like SVM) than it is in applications based on comparison and sorting (like MartiRank).

6.5 Additional Results

As part of our study, we also investigated an anomaly-based intrusion detection system called PAYL [36]. For PAYL, which is an example of *unsupervised* machine learning, the training data simply consists of a set of TCP/IP network packets (streams of bytes), without any associated labels or classificiation. During its learning phase, it computes the mean and variance of the byte value distribution for each payload length in order to produce a model of what is considered "normal" network traffic. During the second ("detection") phase, each incoming packet is scanned and its byte value distribution is computed. This new payload distribution is then compared against the model (for that payload length); if the distribution of the new payload is above some threshold of difference from the norm, PAYL flags the packet as anomalous and generates an alert. PAYL may also raise an alert in other circumstances, for instance if the payload length had not been seen in the training data.

In our experiments with PAYL, we created 40 mutants for testing, but only two were killed using our approach. One reason for this poor performance is that we were only able to automate two metamorphic properties for PAYL: permuting the order of the packets in the training data set; and permuting the order of the bytes within the payload ("message") in each packet. Although permutation of the input proved to be an effective technique for detecting defects in SVM and MartiRank, PAYL is purely concerned with distribution of values and not at all with their ordering or relationship to each other; thus, it follows that permuting the input would have very little effectiveness at killing the types of mutants that we introduced.

We mention this result here to demonstrate that metamorphic testing is not a silver bullet for applications that have no test oracle, and that its effectiveness relies heavily on a combination of the types of metamorphic properties, the types of defects being targeted, and the nature of the application itself. Although we have demonstrated effectiveness for some types of defects and some types of applications, further work is required to more precisely categorize the defects and applications for which the approach is most suitable.

7. LIMITATIONS AND FUTURE WORK

Aside from the limitations described above, other challenges remain. Once a defect has been revealed, fault localization techniques need to be used to find the errant code. However, localizing faults during system testing is quite difficult because the fault could come from anywhere in the code. Thus, this particular approach may be more suitable for detecting defects than for localizing them. However, we foresee an approach that combines coarse-grained Automated Metamorphic System Testing with fine-grained testing at the unit level (such as [26]), such that defects detected by the former can be localized with the latter. At this point, though, automated fault localization is outside the scope of work.

Our current implementation of the Amsterdam framework only supports metamorphic properties that deal with comparing the outputs of independent executions in some prescribed way. The framework does not currently support metamorphic properties such as "$ShortestPath(a,\ b) = ShortestPath(a,\ c) + ShortestPath(c,\ b)$ where c is some point in the path from a to b", *i.e.*, properties that depend on the result of the initial execution of the program, since the executions are meant to run in parallel. We leave this as future work.

Another limitation of the framework comes from the various output formats that such applications may produce. For instance, even though many machine learning classification applications use a standard set of input file formats (typically CSV or ARFF), the formats of the outputs vary greatly, since each algorithm represents its model differently. Thus, it will likely be the case that the tester needs to create a custom utility to compare outputs. However, this does not obviate the need for a testing framework, which still provides substantial out-of-the-box functionality with respect to transforming inputs and executing the code, and the notion of a heuristic oracle for use with metamorphic testing.

Additional future work may also include the automatic detection of metamorphic properties, similar to the work that has been done in discovering likely program invariants [11] [15] and algebraic properties [17]. It could be argued that static analysis of the code may be able to determine whether these properties hold, and we have begun preliminary investigations, though such an approach would more likely be useful at the unit level than at the system level, because of the size of the code segments to be analyzed. Further research will be required to determine what are the limits for such approaches when detecting and checking these metamorphic properties.

Finally, further investigation should explore the application of these techniques to other domains of non-testable programs, such as simulation, optimization, and scientific computing.

8. CONCLUSION

In this paper, we have presented an approach called *Automated Metamorphic System Testing*, which addresses some of the limitations of metamorphic testing so that it can be an efficient technique for testing applications that deal with large, complex data sets. We have also presented a tool called *Amsterdam* that facilitates the manner in which Automated Metamorphic System Testing is conducted, and allows metamorphic testing to continue in the deployment environment with minimal impact on the user.

Aside from the approach and the implementation, our contributions also include a variant called *Heuristic Metamorphic Testing* that seeks to reduce the number of false positives and address non-determinism. Last, we have presented the results of empirical studies of various real-world machine learning applications, and demonstrated the effectiveness of our testing approaches. We hope that our work helps to increase the quality of the "non-testable programs" being developed in machine learning and other fields as well.

9. ACKNOWLEDGMENTS

The authors would like to thank T.Y. Chen and Sal Stolfo for their guidance and assistance. The authors are members of the Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1.

10. REFERENCES

[1] SVM application list. http://www.clopinet.com/isabelle/Projects/SVM/applist.html.

[2] J. R. Abrial. *Specification Language Z*. Oxford Univ Press, 1980.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc of the 27th International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

[4] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, Jan. 1995.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc of the 2002 ACM SIGSOFT international symposium on software testing and analysis*, pages 123–133, 2002.

[6] T. Y. Chen, S. C. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Dept. of Computer Science, Hong Kong Univ. of Science and Technology, 1998.

[7] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 44(15):923–931, 2002.

[8] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proc. of the ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*, pages 191–195, 2002.

[9] W. J. Cody Jr. and W. Waite. *Software Manual for the Elementary Functions*. Prentice Hall, 1980.

[10] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *Proc. of the ACM '81 Conference*, pages 254–257, 1981.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely programming invariants to support program evolution. In *Proc. of the 21st International Conference on Software Engineering (ICSE)*, pages 213–224, 1999.

[12] A. Gotleib and B. Botella. Automated metamorphic testing. In *Proc. of 27th annual international computer software and applications conference (COMPSAC)*, pages 34–40, 2003.

[13] P. Gross et al. Predicting electricity distribution feeder failures using machine learning susceptibility analysis. In *Proc. of the 18th Conference on Innovative Applications in Artificial Intelligence*, 2006.

[14] R. Guderlei and J. Mayer. Statistical metamorphic testing - testing programs with random output by means of statistical hypothesis tests and metamorphic testing. In *Proc of the Seventh International Conference on Quality Software*, pages 404–409, 2007.

[15] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*, pages 291–301, 2002.

[16] J. A. Hanley and B. J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143:29–36, 1982.

[17] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. of the 17th European Conference on Object-Oriented Programming ECOOP*, 2003.

[18] D. Hoffman. Heuristic test oracles. *Software Testing and Quality Engineering*, pages 29–32, 1999.

[19] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[20] J. Knight and N. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[21] T. Mitchell. *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, 1983.

[22] C. Murphy and G. Kaiser. Improving the dependability of machine learning applications. Technical Report CUCS-49-08, Dept. of Computer Science, Columbia University, 2008.

[23] C. Murphy, G. Kaiser, and M. Arias. Parameterizing random test data according to equivalence classes. In *Proc of the 2nd international workshop on random testing*, pages 38–41, 2007.

[24] C. Murphy, G. Kaiser, M. Chu, and I. Vo. Quality assurance of software applications using the in vivo testing approach. In *Proc of the Second IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2009.

[25] C. Murphy, G. Kaiser, L. Hu, and L. Wu. Properties of machine learning applications for use in metamorphic testing. In *Proc. of the 20th international conference on software engineering and knowledge engineering (SEKE)*, pages 867–872, 2008.

[26] C. Murphy, K. Shen, and G. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Proc of the Second IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2009.

[27] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. UCI repository of machine learning databases. University of California, Dept of Information and Computer Science, 1998.

[28] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. S. Reis. Checking the conformance of Java classes against algebraic specifications. In *In Proceedings of ICFEM'06, volume 4260 of LNCS*, pages 494–513. Springer-Verlag, 2006.

[29] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proc of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 361–376, 2002.

[30] J. Platt. Fast training of support vector machines using sequential minimal optimization. *Advances in Kernel Methods, Support Vector Learning*, 1999.

[31] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1993.

[32] S. Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of the 1991 international symposium on software testing, analysis, and verification*, pages 123–129, 1991.

[33] S. Sankar, A. Goyal, and P. Sikchi. Software testing using algebraic specification based test oracles. Technical Report CSL-TR-93-566, Dept. of Computer Science, Stanford Univ., 2003.

[34] C. Spearman. Footrule for measuring correlation. *British Journal of Psychology*, 2:89–108, June 1906.

[35] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[36] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *Proc. of the Seventh International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2004.

[37] E. J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, November 1982.

[38] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, 2nd Edition*. Morgan Kaufmann, 2005.

[39] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen. Improving the quality of computational science software by using metamorphic relations to test machine learning applications. Technical Report CUCS-004-09, Dept. of Computer Science, Columbia University, January 2009.